A Programmer's Guide to FSMGDOS

Introduction

The latest incarnation of GDOS is called FSMGDOS which features a powerful outline font generator. FSMGDOS allows you to scale fonts, as well as rotate them. Programming with FSMGDOS is a fairly easy task. FSMGDOS was written as a superset of the VDI specifications, so that a minimal amount of changes to existing programs are needed in order to take advantage of the new features. In fact, most GDOS applications require no changes to use FSMGDOS, although they will not take advantage of all of its features. Anyone with experience programming with GDOS should find the new features to be easy-to-use, logical extensions to the original format. This document discusses the new concepts and features found in FSMGDOS, that will allow the programmer to produce a fast, powerful, and memory-efficient application. Please note that FSMGDOS comes with new drivers. These drivers are required for FSMGDOS to work on printing devices.

Compatibility and Fixed Point Sizes

One of the great advantages of having FSMGDOS is being able to arbitrarily scale fonts. New calls have been introduced to provide easy access to the outline fonts, so that programs can create and use any size font. Of course, with the introduction of new calls, existing GDOS applications had to have a way of using the new fonts. A lot of effort was made so that FSMGDOS would be compatible with existing programs, and as a result, applications can use existing GDOS calls to get at the outline fonts. Unfortunately, certain limitations were imposed in order to achieve this compatibility.

When using existing GDOS calls, FSMGDOS requires the user to select a fixed set of point sizes that he/she wants to use. Users either have to edit the "extend.sys" file (an ASCII configuration file, similar to the "assign.sys") or use a desk acessory to add or delete point sizes for FSM fonts. Therefore, GDOS applications that were written for GDOS 1.1 can access FSM fonts from the fixed set of point sizes (Note: unlike bitmap fonts, point size doubling is not available for FSM fonts). This feature of making the user specify a fixed set of point sizes is meant to provide compatibility with old applications, and should not be used when writing new or updating existing software. The user no longer has to limit the sizes of the fonts, since FSMGDOS has a new set of calls that can create any size of font. These calls (discussed in a section below) should be used whenever outline fonts are used.

NOTE: To access an FSM font, the extend.sys file must specifiy at least one point size for every font in the main list.

Identifying FSMGDOS

An application can find out whether or not FSMGDOS is running by making a trap 2 call with -2 in d0. If FSMGDOS is installed, d0 will contain '_FSM' (0x5F46534D) in d0.

Am I an Outline Font?

With bitmap fonts, applications used to go through a sequence to find out what point sizes were available for a particular font. Since FSM fonts are arbitrarily scalable, applications should check

to see if a font is an outline font, and bypass the point size inquiry if the font is scalable. The easiest way for an application to tell whether or not a font is arbitrarily scalable is to use the vqt_name() call. When the vqt_name() call is made, a flag is set to 1 if the font in question is an outline font (0, otherwise). The flag is found in the 33rd element of the "name" array that contains the name of the font. Note that the flag was placed in the name array to conserve compatibility with older bindings to the vqt_name() call.

Users of this new feature must use a new binding for vqt_name().

Setting Point Sizes

There is a new call for setting the size of an outline font. It is called vst_arbpt(), "set arbitrary point size". The call is used to set the point size of the current font. Note that new applications should not use the vst_point() call to set the point size for FSM fonts; vst_point() will only access the fixed point sizes specified in the "extend.sys" file, and is actually slower than vst_arbpt(). Like vst_point(), vst_arbpt() takes a point size as its input, and should return the same point size as the value which was input. For more information, see the attached bindings.

The other way to set the height of an outline font is to use the vst_height() call. This is not a new call, but a vst_height() call on an FSM font will yield height information that matches the input. In other words, if an application asks for a 99 pixel high font, it will get a 99 pixel high font (for a bitmap font, GDOS will scale as best as it can, although the characters tend to be very jagged). If necessary, the value returned by the call can be converted into points, by the standard conversion using 72 points/inch and the device dots/inch (Note that this conversion is only approximate as a 72 point font is not one inch in all faces).

Output Differences

One of the major differences in FSMGDOS is quite subtle. A limitation of the bitmap font format is that characters are contained in cells which are placed side-by-side to form a string. In FSMGDOS, outline font characters are not forced into cells, and can be overlapped. As a result, spacing (or placement) of characters becomes more aesthetically pleasing.

Due to this characteristic, outline font characters look best when written in transparent mode. In replace mode, the background rectangle is blitted, then the character bitmap is written in transparent mode. If a string is printed character-by-character, parts or fringes may be erased. So, to avoid incomplete output, either the application should use replace mode only to print whole strings (letting FSMGDOS clear the background) or it can use transparent mode and make sure that the background has been cleared by itself.

Reverse transparent mode output is very different as well. Due to the nature of FSM text, reverse transparent text output becomes "blocky" and aesthetically displeasing. Furthermore, for similar reasons, rotated text in this mode tends to overlap on itself. It is not recommended that programs use this mode.

WYSIWIG

WYSIWYG (What You See Is What You Get) is the goal of all graphics-based document production systems. An outline font system is ideal for basing such a system. Fonts are created with outlines: A description of the lines that enscribe the outlines of characters is mathematically scaled to correspond to the desired point size of a particular dot density. Then the outline is used as kind of a stencil. The outline (or stencil) is placed on top of a grid that correspond to

the pixels of a device (e.g. the screen), and the pixels within the outline are turned on. Hence, a bitmap is generated of a character in a point size of a particular dpi (dots per inch). The same outline can be scaled to correspond to the pixels of any device, such as a laser printer, and the resulting character should be the same size as the character on screen. Thus, an application can display WYSIWYG (i.e. the screen output matches the printer output in size and proportion).

Unfortunately, this is not entirely true. Obviously, the screen and printer can differ wildly with respect to resolution. For example, ST high resolution has a dpi of 91 and the SLM series laser printers have a dpi of 300. Text of the same point size should indeed be the same size, but the screen characters will be much "blockier" or more granular. Of course, this is perfectly natural. What complicates the problem is the fact that the device resolution can become so granular that not only is the aesthetics of the character affected, but the actual metrics of a font as well.

In other words, the values that describe the height, width, and placement of the character can become inaccurate due to the loss of precision at low resolution. Again, this inaccuracy is to be expected; there is no such thing as a fraction of a pixel. For example, take the character 'm'. The outline of 'm' is scaled so that it is 12 points on the screen. There are three vertical lines on an 'm', and let's suppose that the scaling says that the two outside lines of the 'm' are lines that are two pixels wide, but the middle one is slightly smaller and should be 1.5 pixels wide. How many pixels should the middle line be? If FSMGDOS decided to round, the line would be two pixels. On the other hand, it could just truncate the value and make the line one pixel wide. Consider both possibilities: what should the width of the character be? Unless some other compensating factor is used (such as making the distance between the lines a pixel shorter or longer, as the case may be), the width of the 'm', in both cases, is half a pixel off. Therefore, a string of ten 'm's could be five pixels longer or shorter than what the scaler says the length of those charaters should be. Furthermore, WYSIWYG is no longer valid, because the printer has a different resolution which would yield a different error. In the case of the SLM series laser printer, the resolution is much greater than the screen, making the possible error much less significant.

FSMGDOS deals with the above problem in one of two ways. The easiest way is to ignore the problem. Ignoring the problem might not seem to be a viable solution, but as long as FSM mathematically rounds values consistently, output will be predictable (somewhat like bitmap fonts), although not precisely WYSIWYG (In fact, FSM font text will be further from WYSIWYG than bitmap fonts, since bitmap fonts were created by hand). The other method is to allow applications to deal with the fractions themselves. The following sections will elaborate on how to create the most accurate output that FSMGDOS is able to create.

Character Placement

In the old GDOS, character placement was very simple. If one knew the width of a character, one knew exactly where the next character should be placed. A character was always contained in a cell of fixed width, so that the next character was placed right after the cell, offset by however many pixels the width was. In FSMGDOS, as explained in a previous section, characters can overlap. Therefore, character placement cannot be made in terms of character width. A new call has been installed in GDOS that works like vqt_width(), but returns a vector for where and how far to advance the cursor. The vector returned specifies the arbitrary point where one character ends and the other begins. This is especially useful with rotated characters, since vqt_width() only tells you how far to advance in the x direction. The call is called

vqt_advance(), which returns the vector that when added to the last cursor coordinate, yields the next cursor position.

Vqt_advance() returns four integers: the x-advance, y-advance, x-remainder, and y-remainder. The advance values are a vector that yields the next cursor position. The remainders are just the fractional portion of the advance values. To get the most accurate output, FSMGDOS uses the remainders when calculating the next postion of a character in a string. A special text call, v_ftext(), is used as a substitute for v_gtext() that uses the remainders when outputting text. It works exactly like v_gtext(), except that the placement of characters is slightly different. In addition, vqt_ fextent() has been provided for the v_ftext() call to behave in the same manner as vqt_extent() does for v_gtext(). V_gtext(), vqt_extent(), and vqt_width() behave as if the remainders are always zero.

For both v_gtext() and v_ftext(), the notion of character width must be redefined. Vqt_width() no longer specifies the cell width as defined in older documentation. For FSM fonts, the character width should be interpreted as the number of pixels to advance in the x-direction to place the cursor **for the zero degree case and for v_gtext() only.** This limited definition on vqt_width() is used for compatibility purposes; existing applications that use v_gtext() and vqt_width() can freely use FSM fonts, just like bitmap fonts. Furthermore, FSM font characters, unlike the old-style bitmap fonts, can go beyond the left and right boundaries imposed by vqt_width() (and vqt_advance()).

The slightly different interpretation of character width also affects the vqt_extent() and vqt_fextent() call. With bitmap fonts, the vqt_extent() call returned the coordinates of a rectangle that enclosed the given text string. Unfortunately, the call was based on the fact that vqt_width() returned a value that spanned the width of an entire character. Since, the return value is no longer guaranteed to be as wide as the character, the vqt_extent() call might not enclose the string. Of course, most strings will be entirely enclosed, but many will go out of the defined bounds. This is especially common with any slanted fonts such as Lucida Calligraphy or Zapf Chancery. The problem is further complicated when text is rotated (See the next section for details).

The way to solve the problem for both rotated and non-rotated text is to make the width of the box returned by vqt_extent wider. More precisely, the maximum character width (of the font) should be added to both left and right sides of the extent box (in the case of rotated text, the width should be extended in relation to the baseline). The resultant box should encompass all of the text string, and can be used as a blit buffer or as an extent box for erasure.

Note: The maximum character width returned by vqt_fontinfo() tends to be very large. For purposes of adjusting the extent box, if the width is too large and the characters in the box are normal characters (e.g. 'a' - 'z', 'A' - 'Z', numbers), the width of a capital 'M' or 'W' can be used as an approximation.

Another placement problem may occur because of the vertical alignment values. The ascent, descent, and half lines should be interpreted differently for bitmap fonts. For bitmap fonts, more often than not, the ascent line was equal to the top, and the descent was equal to the bottom. Programs could also assume by convention that lower case letters would stay below the half line, and descenders would stay above the descent line. With FSMGDOS, this is no longer the case. The aforementioned vertical alignment values are approximates, and many characters will not

stay within these boundaries. This fact should not affect many programs. Note that top and bottom are specified such that there is much more whitespace above and below a character. Many applications used these values to position lines vertically. With FSM characters, one might wish to reconsider using top and bottom, since line spacing will be noticably larger than with bitmap fonts.

Using Rotation

Incorporating font rotation is not as simple as incorporating font scaling. A certain number or combination of features on bitmap fonts do not work for rotated fonts, due to practical considerations. For example, justified text does not work with angles other than multiples of 90 degrees. Note that rotation works for the 90 degree cases, but one should watch for the 90 and 270 degree cases which, in many cases, are affected by non-square aspect ratios. In the case of bitmap fonts, screen or printer drivers do not account for aspect ratios.

When using arbitrary rotation, it is strongly recommended that applications use v_ftext(), vqt_advance(), and vqt_f_extent(). The remainders used in these calls are crucial to having correct output on shallow angles.

Also, vertical and horizontal alignment should be accounted for when rotating text. The alignment affects the point at which the text is rotated.

For vqt_width(), FSMGDOS will return the x-component of the advance vector in all rotations except when the rotation is 90, 180 or 270 degrees. In other words, at 90 degree rotations, vqt_width() will behave like bitmap fonts are being used, but at non-90 degree rotations, the x-component of the advance vector is used. Programs that need to keep track of widths in the "older" sense (i.e. zero degree rotated) should set the rotation to zero, then inquire the widths for the character set.

For vqt_extent(), FSMGDOS will return a rectangle with overlap characteristics similar to the zero-degree case. The way vqt_extent() works is that it gets the extent rectangle for the zero-degree case, then GDOS rotates the points of the rectangle. Therefore, the rectangle will be exactly the same in dimensions and text placement as the non-rotated case (ignoring mathematical/low resolution aberrations). Note that the rectangle will touch both axes, preserving this characteristic as well.

When using vqt_extent() to determine the size of a temporary buffer for faster blits or text erasure, one should add the maximum character width to both ends of the baseline to extend the original extent box (the extent box should have its baseline increased by two times the maximum character width). Then, a non-rotated box needs to be determined to encompass the extent (for rectangular blits).

Special Effects

Special effects are the built-in algorithms that generate lightening, bold, skewing, and outline on fonts. It is recommended that wherever possible, the actual font style be used instead of the algorithm (e.g. use Lucida Bold instead of using thickening with Lucida Roman), since the actual font style will be generated from its own font file. Such cases will yield much better output if the actual style/point size is used. Of course, not all of these effects are available as a separate font style. Also note that a new skew call is available (see below).

An issue directly related to special effects is the scratch buffer. The scratch buffer is the memory allocated for the effects' algorithm; the size of the memory is based on the largest point size of the largest font in the "extend.sys" or "assign.sys" file. Due to the nature of arbitrarily scaled fonts, the buffer cannot possibly be big enough (since the buffer's size is based on the dimensions of the largest available point size of all available fonts). Therefore, if one wishes to use special effects at all, one must make sure that the buffer is big enough. The way to ensure that the size is big enough is to find out what the biggest point size is for all fonts in the "extend.sys" or "assign.sys". Then determine what size is being requested (by comparing the point sizes). If the requested size is bigger, the application should not allow the special effects to be used.

Another problem arises with special effects and rotation. Some special effects made available through the vst_effects() call don't work very well with FSMGDOS. In particular, bolding and skewing don't have the desired effects on text when rotated. This is because these effects are algorithmically generated, and the algorithms only work for non-rotated text. If the effects are used on rotated FSM text, the results may look rather odd (e.g. distorted characters, crooked strings of characters). Special effects should only be used if you have checked to see that the output is what you want.

Memory Management

Memory management is simple, since FSMGDOS does all of the work. An application only has to worry about whether or not it has enough room to run. The user must worry about if s/he has enough memory for FSMGDOS. The user has access to two variables, the size of the cache and how to divide it. Once set, two caches are allocated whose total size corresponds to the user-set size and are split according to the percentage supplied by the user (default is a 50-50 split). One cache is devoted to storing the character bitmaps. The other is used for miscellaneous memory needs resulting from bookkeeping and the generation of characters. Applications can find out what the biggest chunk of memory is left for each cache by using the vqt_cachesize() call (see bindings).

The cache for character bitmaps uses a simple cache scheme to reduce the overhead of managing a more sophisticated one. When FSMGDOS determines that there is no more room in the cache, it flushes it. This is done automatically, whenever a request for memory is made by FSMGDOS, and there is no way for the application to anticipate when this will happen. However, a call has been provided so that an application can flush the cache at any time. The call can be used in the following manner: When an application is about to print to a device such as a laser printer, the cache is full of screen fonts. It is unlikely that the cache is full, but when the application goes out to print, a v_flush_cache() will make room for printer-sized fonts. The application could possibly avoid an involuntary flush of the cache which might get rid of characters that it may immediately need. The same reasoning works for the screen. After the printing occurs, there might be a small amount of room to accomodate some of the screen characters (some, not all). So if the cache is not flushed before generating screen characters, the cache might become full before redrawing the screen, thereby generating some screen characters more than once. It would be better if the application cleared the cache for screen characters. Note that the cache can be saved and loaded at appropriate times to further enhance the performance. See the section called "Saving the Cache."

The size of characters are determined by using the vst_arbpt(), vst_height(), or the vst_point() call (see above). If, due to memory limitations, FSMGDOS cannot generate a requested size, the

above calls will not return a smaller size that it can accomodate, like bitmap fonts do. Depending on what error mode has been selected, the error code should be checked, or an error message will appear. Memory requirements for bitmap fonts are fixed and known. With arbitrarily scaled fonts, different characters need different amounts of memory (i.e. some characters are just physically bigger than others), and thus, there is no limit enforced on the maximum point size. For example, an application should not be prevented from using a 100 point comma simply because it cannot accomodate a capital 'w' from the same font. For all point sizes, the vst_height() and vst_arbpt() call will not restrict sizes on FSM fonts, and FSMGDOS will attempt to generate whatever characters that it is asked. Of course, an application could use the vqt_cachesize() call to check to see if there is a big enough chunk of memory. An application would have to approximate the size of the resultant bitmap, and issue a warning (or prevent the user from doing the action). Otherwise, if there is not enough memory, an error message will be output to the screen, telling the user to allocate more memory for the cache and reboot.

Note that the simple cache management scheme was chosen in favor of an LRU algorithm or some more sophisticated algorithm. This was done in consideration of the typical usage of FSM characters. It seems reasonable to assume that the cache will overflow when the printer is being used. Printers tend to be a much higher density medium compared to the screen, and character bitmaps for printer devices will fill up the cache much quicker. In addition, cache sizes will tend to be tuned such that the screen fonts will be small enough to fit within the cache. The garbage collection required by a more sophisticated algorithm will make FSMGDOS' memory management slower and less predictable.

The other cache, the miscellaneous cache, is trickier. It will neither flush itself, nor is there any way for an application to manage its size. The cache is used for many internal data structures for FSM, so it is hard to garbage collect (and it is subject to fragmentation). The only time it is cleaned up is when all of the workstations are closed, and even then, part of the cache is still being used. If the cache is overflowed, FSMGDOS will put up a nasty message stating that the user should allocate more memory and reboot. The best that applications can do to prevent this from happening is to warn the user beforehand. You can anticipate how much memory is needed for some action (see above), and warn/prevent the user from doing it. Of course, this technique is only good for gross errors (e.g. users asking for a 300 point font). Unfortunately, this cache cannot just get more system memory when it runs out.

There is a formula that determines how much miscellaneous cache FSM needs to generate a particular character. This should not be used as an absolute maximum, but it should give you a general idea.

84 * (width + height) = # of bytes

So, for a particular font, using inquire calls to get the form height and the maximum character width, you can find out how much memory the FSM module will need. Unfortunately, the maximum character width is not entirely accurate because the width is interpreted as being the advance vector and bitmaps of characters can be wider (or smaller) than that vector. A decent safety margin is necessary. Note that the memory referred to is from the non-bitmap portion of the FSM cache.

Saving the FSM Bitmap Cache

The FSM bitmap cache can be saved and reloaded at any time. This is primarily useful for users who wish to save previously generated characters so that they can come back to the same environment and not wait for character generation. On the other hand, applications can do several interesting things, such as save caches for the screen just before an application tries to print, so that after printing, the cache can be flushed and reloaded with the screen's old cache. Similarly, the printer cache can be loaded for the next printing, in the case that it is the same or similar document. Applications can also save and manage caches of whole sets of fonts, and load them in when necessary or when the user requests a font.

The commands for cache manipulation are simple. V_savecache() takes the entire content of the cache containing FSM characters and saves them out to disk (to the file and path specified in the parameters). The v_flushcache() call clears out the entire cache. The v_loadcache() routine takes a filename (which may include a path) and extracts the cache entries to place in the cache. In addition, it takes a flag that says whether or not to append the entries to the cache or just flush the cache first.

Please refer to the attached bindings.

New Effects

There are two new effects that can affect the look of FSM fonts. One is vst_skew(). This call allows the application to set the shearing value in the transformation matrix. Note that the shearing can be used in conjuction with rotation. This is the only way to get skewed characters that are rotated. The shear value is taken in degrees which should range from -90 degrees to 90 degrees, although shearing at either end of the range will produce unintelligible text (since a character skewed at 90 degrees is an infinitely long straight line).

The other new call is the vst_setsize() call. This is just another "set point size" call, but it affects the width of characters. This is useful for compressing or expanding text. Note that for compatibility reasons, the setsize is reset whenever the size of the characters are set in the y-direction (i.e. with a vst_point(), vst_arbpt(), or vst_height() call). If the setsize is negative, the text will be mirrored over the y-axis.

Note that another effect can be made by using the vst_arbpt() call. If the point size is negative, characters will be reflected over the x-axis, and a mirroring effect can be achieved.

The Extend.sys

It is generally not recommended that applications manipulate the extend.sys file. FSMGDOS will not behave predictably. On the other hand, it might be useful for you to know what can be changed and when the changes take effect.

In general, if the timestamp on the extend.sys file changes, FSMGDOS will reload the file at a vst_load_font(). This is meant to reflect changes made to the extend.sys **between applications** Note that a vst_load_font() in the middle of an application (for loading printer fonts, for example), can indeed cause a reloading of the extend.sys, and the behavior of FSMGDOS will not be predictable (if the extend.sys has been modified).

Fonts and point sizes should be changed outside of the application (Note that you really should not be using a point size entry, since this is used for backward compatibility puposes. Please use

vst_arbpt() or vst_height().). Also, widthtables should be turned on or off outside of an application. The path to the FSM folder and the symbol/hebrew file specifications can also be altered outside of an application. However, the cache sizes will never be changed unless a system reboot is made.

Please refer to the FSM user's guide or the "extend.sys" document for more information.

Widthtables

Widthtables were created to provide compatibility with older programs. As a side effect, they are useful to speed up applications in certain cases. In particular, if a user builds widthtables for fonts in his/her extend.sys and enters the point sizes for those fonts and turns the widthtables flag on, then inquiry calls for the widths of particular fonts and point sizes will be relatively quick. Otherwise, FSMGDOS would have to calculate the values from scratch, loading and reading from the font file.

Many applications go through the process of filling out their own widthtables by making the vqt_width() calls for all of the characters in all fonts and all point sizes. Although with bitmap fonts, this might have been a faster method of accessing this data, but with FSM fonts, this might not be such a good idea. As mentioned above, this process is very time consuming without having users preset the point sizes that they want to use **and** have them build widthtables for them. The whole process is restrictive and confusing. It is recommended that your applications and any future ones avoid filling out and using your own widthtables. By always using the vqt_width() or vqt_advance() call, FSMGDOS is responsible for caching that data (which it does when a character is generated). Note that if v_ftext() or vqt_f_extent() is used, widthtables are useless, since the tables do not save the remainders (or even the y-advance). Moreover, to speed up your applications, they should only ask for widths/advances when needed, so that they don't inquire on characters that they will never use.

However, FSMGDOS is responsible for caching **screen** font widths only. When outputting to any other device such as a printer, width tables will not be cached. It is suggested that applications cache widths for devices other than the screen. Otherwise, for example, each time a word processor prints a document, it must spend time calculating the widths of different fonts.

Finally, using widthtables, in a general sense, is incompatible with the nature of FSM fonts. Since they are arbitrarily scalable, there is no way of predicting what widthtables will be needed. In other words, users would have to choose the point sizes that they want to use before running a program, while the program is supposed to allow you to choose any size font. In general, widthtables should be avoided when supporting arbitrarily scaled fonts, but in certain specialized cases, you can maximize the performance of FSMGDOS by using them (which is actually the user's responsibility).

Accessing New Characters

FSM fonts contain characters outside of the Atari character set. A call has been provided so that applications can access these characters by careful modification of FSMGDOS-internal data tables. The FSM character set is specified in gascii, while the Atari set is in ascii. There is a table inside of FSMGDOS which maps all ascii values to gascii. If a user/application has particular gascii characters that s/he wants to use and knows that the characters are in a particular FSM font file, the gascii values can be substituted for ones already in the table.

The vqt_get_tables() call retrieves the addresses of two tables. One is the aforementioned gascii table. The other is the style table. Both tables contain the constants used to locate and generate all outline fonts. The style table is the table which tells FSMGDOS which font file contains the requested characters. In other words, the gascii table entry for the character 'a' will contain the gascii value for 'a', and the style table will contain an index that says that that character is in the main font file.

Hence, if there was some character in the music font (e.g. an eigth note) that you could not access, you could use this call to substitute some unused character with the one needed. Then, you must make sure that the entry in the style table refers to the right type of font file (e.g. the eighth note is in the music font, which is a symbol font, so the style table entry must index into the symbol font file).

Note that remapping the gascii and style table as described above can confuse the cache. Although an application should not blow up, any saved and reloaded cache might contain incorrect characters if those characters were remapped by the application (i.e. characters might not be the ones expected by the user). Any remapping should take place as soon as possible, and the cache should be flushed. Furthermore, the widthtables should be considered inaccurate, since they contain widths for characters that may no longer exist or that have been changed.

Error Output

There are currently two modes of error output for FSMGDOS. The default method of reporting errors is printing an error to the screen. This mode is good for older GDOS-compatible programs which were not written to handle FSM errors. The second method involves passing in an address of an integer variable which will be stuffed with an error code if one occurs. The error code variable should be checked whenever a text call is made (e.g. v_gtext(), vst_point(), v_savecache()). Note that the application is responsible for resetting the error code, so therefore, whenever an error is detected, the application should clear the variable before making further calls. See the description of vst_error() below, and the list of calls that could incur an error.

Outlines

Calls have been provided so that applications can get at the actual outlines that FSM generates. The outlines come packaged in a structure that contains specifications for conic splines that define a particular character. When v_getoutline() is called with a character "ch" as a parameter, an address of a structure is returned which contains the outline of character "ch". The structure, called a component, is defined as follows:

```
typedef struct fsm_component_t {
        short   reserved1;
        struct fsm_component_t *nextComponent;
        unsigned char numPoints;
        unsigned char numCurves;
        unsigned char numContours;
        unsigned char reserved2[13];
        fsm_data_fpoint_t *points;
        unsigned char *startPts;
} fsm_component_t;
```

```
typedef struct {
        fsm_fpoint_t pt;
        fsm_fpoint_t cpt;
        fsm_int sharp;
} fsm_data_fpoint_t;

typedef struct {
        fsm_int x, y;
} fsm_fpoint_t;

typedef struct {
        short value, remainder;      /* remainder is mod 16384 */
} fsm_int;
```

Within each **fsm_component_t**, there is a conic spline for each contour. Point (**pt**) and tangent point (**cpt**) coordinates are integer values with remainders; The **sharp** field is the integer value with remainder of the square of the conic sharpness, s**2. **points** is an array of **numPoints** points.

startPts is an array of **numContours** +1 point indices; startPts[n-1] is the point index of the first point of contour n. startPts[numContours] = numPoints+1 and indicates the last point of the last contour. A curve is determined by a positive sharp value; If outline is a structure of type fsm_outline_t, and outline.points[pointNumber-1].sharp is not zero, then the application program will need to use the following in order to build the conic spline:

first end point - outline.points[pointNumber - 1].pt
tangent point - outline.points[pointNumber - 1].cpt
second end point - outline.points[pointNumber].pt
sharpness - sqrt(outline.points[pointNumber - 1].sharp)

To convert conic splines to the beziers available in FSMGDOS, place the first bezier control point on the line connecting the first end point and the tangent point at a ratio **f** of the distance between these points. The corresponding thing is done for the second end point.

f = 4/3 * **s** / (**s**+1)
s = the square root of **sharp**

Line-A
Don't use it. FSMGDOS will never support it. FSM text is not compatible with Line-A. Furthermore, since FSMGDOS caches bitmap text, Line-A can not access outline fonts at all.

Bitmap Font Cache and FONTGDOS
FSMGDOS contains caching facilities for bitmap fonts as well. The cache is entirely separate from FSM font handling. Please refer to the users document for more details.

The difference between FSMGDOS and FONTGDOS is that FONTGDOS has the FSM generation routines removed. It will still parse the "extend.sys", but it takes the "FSMCACHE = " value as the value for a miscellaneous cache for itself (and will use all of the amount towards this cache, as

opposed to just a percentage). Programming FONTGDOS is identical to programming for GDOS 1.1, except that the Bezier features are installed.

Note: In order to determine if FONTGDOS is installed, place a -2 in d0 and make a trap 2 call. D0 will contain '_FNT' upon return.

Calls Affected by Vst_error()

The following is a list of VDI calls that could cause an error in FSMGDOS. Using vst_error(), an application can set the error output mode such that it places an error code into an address provided by the application. Therefore, the application can discover if an error has occured by checking for an error whenever any of the following calls are used:

v_gtext()
v_justified()
vst_point()
vst_height()
vst_font()
vst_arbpt()
vqt_advance()
vst_setsize()
vqt_fontinfo()
vqt_name()
vqt_width()
vqt_extent()
v_opnwk()
v_opnvwk()
vst_load_fonts()
vst_unload_fonts()
v_ftext()
vqt_fextent()

If the error code is non-zero, an error has occured. The application should handle the error appropriately, and reset the error code to 0 (so that new errors may be detected). Note that the error code should be initialized to 0 before any calls are made; this is to insure that the first error is detected.

FSM Error Codes:

0	-	No Error
1	-	Character not found in font
8	-	Error reading file
9	-	Error opening file
10	-	Bad file format
11	-	Out of memory/Cache full
-1	-	Miscellaneous error

Addendum to Inquire Face Name and Index

A new parameter has been added to this function. It is a flag that indicates whether or not the font in question is an FSM font or bitmap font. The flag will be contained in intin[33]; 0 will signify a bitmap font, and 1 will indicate an FSM font.

Input	contrl(0)	--	Opcode = 130
	contrl(1)	--	Number of input vertices = 0
	contrl(3)	--	Length of intin array = 1
	contrl(6)	--	Device handle
	intin(0)	--	Element number.
Output	contrl(2)	--	Number of output vertices = 0
	contrl(4)	--	Length of intout array = 34
	intout(0)	--	ID Number
	intout(1) to intout(32)	--	32 ADE
	intout(33)	--	flag (1 if FSM, 0 otherwise)

C BINDING

Procedure Name

index = vqt_name(handle,element_num,name)

Data Types

```
int     index;
int     handle;
int     element_num;
char    name[33];
```

Input Arguments

handle = contrl[6]
element_num = intin[0]

Output Arguments

index = intout[0]
name[0] = intout[1]
.
.
.
name[31] = intout[32]
name[32] = intout[33]

Note: To preserve compatibility, the fsmflag is placed in the 33nd byte of the name array.

Get FSM Bitmap Information

This function retrieves placement information for FSM-generated characters. An address of a structure 52 words long is passed in intin[1]. The structure, infoarray, will contain:

infoarray[0] = advance in the x direction
infoarray[1] = remainder of x advance
infoarray[2] = advance in the y direction
infoarray[3] = remainder of y advance

(Note: the remainder is in the same format as the remainders from the vqt_advance() call.)

infoarray[4] = bitmap width
infoarray[5] = bitmap height
infoarray[6-7] = the x offset (real)
infoarray[8-9] = the y offset (real)
...
...

infoarray[46] = bitmap width
infoarray[47] = bitmap height
infoarray[48-49] = the x offset (real)
infoarray[50-51] = the y offset (real)

The first four integers are the advance vector. Following that vector is the placement information. When FSM generates a character, a bitmap of the character is produced. The dimensions of that bitmap are placed in the following two integers of the array. The next two values are the placement vector that goes from the current point (i.e. the current cursor position, assuming baseline alignment) to the upper left hand corner of the bitmap. Since characters can consist of more than one bitmap (e.g. ü), up to seven more sets of dimensions and offsets can follow. Of course, one will rarely see characters with more than three parts. To signify that there are no more parts, the width and height values will contain -1.

The placement offsets are given as reals. Internally in FSMGDOS, negative multiples of 1/2 are rounded down (e.g. -1.5 becomes -1 and -1.6 becomes -2). Therefore, in order to find out where the bitmap has been blitted, the offsets must be converted in the same manner.

Input

	contrl(0)	--	Opcode = 239
	contrl(1)	--	Number of input vertices = 0
	contrl(3)	--	Length of intin array = 3
	contrl(6)	--	Device handle
	intin(0)	--	character
	intin(1-2)	--	address of structure
Output			
	contrl(4)	--	Length of intout array = 0

C BINDING

Procedure Name v_getbitmap_info(handle,ch,infoarray)

Data Types

```
int     handle;
int     ch;
int     infoarray[52];
```

Input Arguments

handle = contrl[6]
ch = intin[0]
infoarray = intin[1][2]

Inquire fsm text extent

This function works exactly like vqt_extent(), but the rectangle returned will account for the remainder values of vqt_advance().

Input

contrl(0)	—	Opcode = 240
contrl(1)	—	Number of input vertices = 0
contrl(3)	—	Length of string
contrl(6)	—	Device handle
intin(0)	—	Character string in current character set.

Output

contrl(2)	—	Number of output vertices = 4
contrl(4)	—	Length of intout array = 0
ptsout	—	same as vqt_extent()

C BINDING

Procedure Name vqt_f_extent(handle,string, extent)

Data Types

```
int     handle;
int     extent[8];
char    string[];
```

Input Arguments

handle = contrl[6]
string = intin

Output Arguments

extent[0] = ptsout[0]

.

extent[7] = ptsout[7]

FSM text

This function works exactly like v_gtext(), but the text returned will account for the remainder values of vqt_advance(). In other words, the text spacing will be more accurate.

Input

contrl(0)	—	Opcode = 241
contrl(1)	—	Number of input vertices = 1
contrl(3)	—	Length of string
contrl(6)	—	Device handle
intin(0)	—	Character string in current character set.
ptsin(0)	—	x-coordinate
ptsin(1)	—	y-coordinate

Output

contrl(2)	—	Number of output vertices = 0
contrl(4)	—	Length of intout array = 0

C BINDING

Procedure Name v_ftext(handle,x,y,string)

Data Types

```
int     handle;
int     x, y;
char    string[];
```

Input Arguments

handle = contrl[6]
string = intin
x = ptsin[0]
y = ptsin[1]

Kill FSM outline

This function takes the address of an outline component generated by v_getoutline() and frees up the associated memory of that outline. The address is passed in intin[0],[1]. Applications should always use this function when they are finished with the outline, or they will run out of memory.

Input

contrl(0)	—	Opcode = 242
contrl(1)	—	Number of input vertices = 0
contrl(3)	—	Length of intin array = 2
contrl(6)	—	Device handle
intin(0-1)	—	address of outline component

Output

contrl(2)	—	Number of output vertices = 1
contrl(4)	—	Length of intout array = 0

C BINDING

Procedure Name v_killoutline(handle,&component)

Data Types

```
int     handle;
long    component;
```

Input Arguments

handle = contrl[6]
&component = intin[0-1]

Get FSM outline

This function generates an outline of the character specified in intin[0], and places the address of the outline in the address passed in intin[1], [2].

Input

contrl(0)	--	Opcode = 243
contrl(1)	--	Number of input vertices = 0
contrl(3)	--	Length of intin array = 3
contrl(6)	--	Device handle
intin(0)	--	character requested
intin(1-2)	--	address of component structure

Output

contrl(2)	--	Number of output vertices = 1
contrl(4)	--	Length of intout array = 0

C BINDING

Procedure Name v_getoutline(handle,ch,&component)

Data Types

```
int     handle;
int     ch;
long    component;
```

Input Arguments

handle = contrl[6]
ch = intin[0]
&component = intin[1],[2]

Set scratch buffer allocation mode

This function sets the method of memory allocation for the scratch buffer. The scratch buffer is memory used to create text with special effects, and it's size is determined by the maximum dimensions of a font. Since FSM fonts can be scaled to any size, the scratch buffer size will not have limits. Furthermore, many FSM fonts don't need special effects (or a scratch buffer), because of actual fonts that correspond to the effects. By default, the allocation mode is 0, which takes FSM fonts into account when calculating the scratch buffer size. If the mode is 1, the size will not be affected by FSM fonts, and will take only bitmap fonts into account. In this case, special effects should not be used for FSM fonts. If set to 2, no scratch buffer will be allocated, and special effects should not be used at all.

Input

contrl(0)	—	Opcode = 244
contrl(1)	—	Number of input vertices = 0
contrl(3)	—	Length of intin array = 1
contrl(6)	—	Device handle
intin(0)	—	allocation mode

Output

contrl(2)	—	Number of output vertices = 1
contrl(4)	—	Length of intout array = 0

C BINDING

Procedure Name vst_scratch(handle,mode)

Data Types

```
int     handle;
int     mode;
```

Input Arguments

handle = contrl[6]
mode = intin[0]

Set FSM error mode

This function sets the mode of error reportage. By default, the mode is set to 1, which means that any FSM errors will be printed on screen. If the mode is set to 0, FSMGDOS will place an error code into the address passed in intin[1], whenever an error occurs. Applications should check this address for errors when making text calls (e.g. v_gtext(), vst_point(), vst_arbpt()). Please see the error code list for specific errors. Note that the address should be initialized to 0 and reset whenever an error occurs.

Input

contrl(0)	--	Opcode = 245
contrl(1)	--	Number of input vertices = 0
contrl(3)	--	Length of intin array = 3
contrl(6)	--	Device handle
intin(0)	--	error mode
intin(1-2)	--	address of an integer variable

Output

contrl(2)	--	Number of output vertices = 1
contrl(4)	--	Length of intout array = 0

C BINDING

Procedure Name vst_error(handle,mode,&errorcode)

Data Types

```
int     handle;
int     mode;
int     errorcode;
```

Input Arguments

handle = contrl[6]
mode = intin[0]
&errorcode = intin[1]

Set character cell height by arbitrary points

This function sets the current graphic text character height in printer points. Unlike the vst_point call, vst_arbpt allows the user to scale to any point size regardless of what is contained in the extend.sys. Note that this call will only work with fsm outline fonts.

Input

contrl(0)	--	Opcode = 246
contrl(1)	--	Number of input vertices = 0
contrl(3)	--	Length of intin array = 1
contrl(6)	--	Device handle
intin(0)	--	Cell height in arbitrary points

Output

contrl(2)	--	Number of output vertices = 2
contrl(4)	--	Length of intout array = 1
ptsout(0)	--	Character width selected in RC units
ptsout(1)	--	Character height selected in RC units
ptsout(2)	--	Character cell width
ptsout(3)	--	Character cell height

C BINDING

Procedure Name

```
set_point = vst_arbpt(handle, point,
                      &chwd,&chht,&cellwd, &cellht)
```

Data Types

```
int     set_point;
int     handle;
int     point;
int     chwd;
int     chht;
int     cellwd;
int     cellht;
```

Input Arguments

```
handle = contrl[6]
point = intin[0]
```

Output Arguments

```
set_point = intout[0]
chwd = ptsout[0]
chht = ptsout[1]
cellwd = ptsout[2]
cellht = ptsout[3]
```

Inquire fsm text advance placement vector

This function returns the x and y offsets which are needed to place the next character of a string in the proper position. This call is necessary when laying down text at rotations other than 0, 90, and 270. In addition, the call returns remainder values for the x and y offsets (mod 16000), so that cursor placement can be calculated for v_ftext() and vqt_f_extent().

Input

contrl(0)	--	Opcode = 247
contrl(1)	--	Number of input vertices = 0
contrl(3)	--	Length of intin array = 1
contrl(6)	--	Device handle
intin(0)	--	Character value

Output

contrl(2)	--	Number of output vertices = 1
contrl(4)	--	Length of intout array = 0
ptsout(0)	--	X advance
ptsout(1)	--	Y advance
ptsout(2)	--	X remainder (mod 16384)
ptsout(3)	--	Y remainder (mod 16384)

C BINDING

Procedure Name vqt_advance(handle,ch,&advx,&advy,&xrem,&yrem)

Data Types

```
int     handle;
int     ch;
int     advx;
int     advy;
int     xrem, yrem;
```

Input Arguments

handle = contrl[6]
ch = intin[0]

Output Arguments

advx = ptsout[0]
advy = ptsout[1]
xrem = ptsout[2]
yrem = ptsout[3]

Inquire device status information

This function takes a device id number as a parameter and reports back to the application whether or not the driver for that device has been installed by gdos. If the driver has been installed, the name of the driver is returned to the application.

Input	contrl(0)	—	Opcode = 248
	contrl(1)	—	Number of input vertices = 0
	contrl(3)	—	Length of intin array = 1
	contrl(6)	—	Device handle
	intin(0)	—	Device id number
Output	contrl(2)	—	Number of output vertices = 1
	contrl(4)	—	Length of intout array = number of characters
	ptsout(0)	—	1 = Device installed. 0 = Device not installed

C BINDING

Procedure Name
vqt_devinfo(handle,devnum,&devexists,devstr)

Data Types

```
int     handle;
int     devnum;
int     devexists;
char    devstr[];
```

Input Arguments

handle = contrl[6]
devnum = intin[0]

Output Arguments

devexists = ptsout[0];
devstr = intout;

Save FSM cache to disk

This function saves the contents of the FSM cache to disk. The function takes a filename as its parameter, and the cache is saved under that filename. The file is created in the current directory.

Input

contrl(0)	—	Opcode = 249
contrl(1)	—	Number of input vertices = 0
contrl(3)	—	Length of intin array = n
contrl(6)	—	Device handle
intin	—	Character string of the filename as ASCII codes in 16-bit words.

Output

intout(0)	—	zero or -1 if an error has occured

C BINDING

Procedure Name

```
ret_val = v_savecache(handle, filename);
```

Data Types

```
int     handle;
char    filename[n];
```

Input Arguments

```
handle = contrl[6]
filename = intin
```

Output Arguments

```
ret_val = intout[0]
```

Load FSM cache

This function loads in the contents of an FSM cache from disk. The function takes a filename and a mode as parameters. The filename specifies what file to open in the current directory. The mode specifies whether or not to append or create a new cache. If the mode is 0, the cache from disk will be appended to the current cache; if it is 1, then the cache will be flushed, and a new cache will be loaded.

Input

contrl(0)	--	Opcode = 250
contrl(1)	--	Number of input vertices = 0
contrl(3)	--	Length of intin array = 1+ n
contrl(6)	--	Device handle
intin(0)	--	mode 0 = append 1 = flush and replace cache
intin(1)	--	First character of filename.
		.
		.
		.
intin(n)	--	Last character of filename
intin	--	Character string of the filename as ASCII codes in 16-bit words.

Output

intout(0)	--	zero or -1 if an error has occured

C BINDING

Procedure Name
ret_val = v_loadcache(handle, filename, mode);

Data Types

```
int     handle;
char    filename[n];
int     mode
```

Input Arguments

handle = contrl[6]
mode = intin(0)
filename(n) = intin(n+1)

Output Arguments

ret_val = intout[0]

Flush FSM cache

This function flushes the contents of the FSM cache. Note that this only flushes the portion of the cache that contains bitmaps of FSM characters.

Input

contrl(0)	—	Opcode = 251
contrl(1)	—	Number of input vertices = 0
contrl(6)	—	Device handle

Output

intout(0)	—	zero or -1 if an error has occured

C BINDING

Procedure Name ret_val = v_flushcache(handle);

Data Types

int handle;

Input Arguments handle = contrl[6]

Output Arguments ret_val = intout[0]

Set character cell width by arbitrary points

This function sets the current graphic text character width (set size) in printer points. An arbitrary set size may be entered to represent the character width. It should be noted that the next call to vst_point, vst_arbpt or vst_height will cancel out this call and will set the set size to be equal to the requested point size.This call will only work with fsm outline fonts.

Input

contrl(0)	--	Opcode = 252
contrl(1)	--	Number of input vertices = 0
contrl(3)	--	Length of intin array = 1
contrl(6)	--	Device handle
intin(0)	--	Cell width in arbitrary points

Output

contrl(2)	--	Number of output vertices = 2
contrl(4)	--	Length of intout array = 1
ptsout(0)	--	Character width selected in RC units
ptsout(1)	--	Character height selected in RC units
ptsout(2)	--	Character cell width
ptsout(3)	--	Character cell height

C BINDING

Procedure Name

```
set_width  = vst_setsize(handle, point,
                         &chwd,&chht,&cellwd, &cellht)
```

Data Types

```
int     set_width;
int     handle;
int     point;
int     chwd;
int     chht;
int     cellwd;
int     cellht;
```

Input Arguments

```
handle = contrl[6]
point = intin[0]
```

Output Arguments

```
set_width = intout[0]
chwd = ptsout[0]
chht = ptsout[1]
cellwd = ptsout[2]
cellht = ptsout[3]
```

Set FSM skew

This call requests fsmgdos to generate skewed characters. It works independantly from the skewed characters generated with the vst_effects call. The requested skew value is represented in tenths of degrees and should be a number between 900 and -900 (plus or minus 90 degrees). Negative degrees will cause characters to "lean" to the left while positive numbers will lean to the right. Note that characters will degenerate badly when their skew approaches 90 degrees. This call only works with FSM fonts.

contrl(0)	--	Opcode = 253
contrl(1)	--	Number of input vertices = 0
contrl(3)	--	Length of intin array = 1
contrl(6)	--	Device handle
intin(0)	--	Requested skew value.
intout(0)	--	Amount of skew

C BINDING

Procedure Name

set_skew = vst_skew(handle, skew)

Data Types

```
int     set_skew;
int     handle;
int     skew;
```

Input Arguments

handle = contrl[6]
skew = intin[0]

Output Arguments

set_skew = intout[0]

Get FSM
Gascii Tables

This call returns the addresses of two tables, which are used internally by fsmgdos. When ascii characters are passed into v_gtext, they are first translated into a 2 byte word which is recognized by the fsm character generator. These translations are contained in the gascii table. This table contains 224 entries with the first entry being the translation for character 32, the second for character 33....etc.
The second table, the style table, gives the caller information on which font file the character was generated from. Recall from previous documentation that the Atari character set is generated from information contained in three fonts, the main font, the symbol font, and the hebrew font. When a character is to be generated, fsmgdos checks this table to determine where to get the character information. The table values are:

0 - From main font file
1 - From symbol font file
2 - From hebrew font file

Why are these tables useful? Fsm fonts have the information to generate many characters which are not in the atari character set. By remapping these gascii values the caller will be able to access these characters.

contrl(0)	--	Opcode = 254
contrl(1)	--	Number of input vertices = 0
contrl(3)	--	Length of intin array = 0
contrl(6)	--	Device handle
intout(0) intout(1)	--	Address of gascii table
intout(2) intout(3)	--	Address of style table

C BINDING

Procedure Name vqt_get_tables(handle, gascii, style)

Data Types

```
int     handle;
long    *gascii;
long    *style;
```

Input Arguments handle = contrl[6]

Output Arguments

```
*gascii =    intout[0], intout[1]
*style =     intout[2], intout[3]
```

Get FSM Cache Size

Returns the largest block size available in each of the two fsm caches. This call can be used to estimate how big a character fsmgdos can handle when it prints a character. A zero (0) in intin[0] will instruct fsmgdos to return the largest allocatable block in the character bitmap cache. A one (1) will return the same information for the data structure cache. Please refer to the section on memory management for more information on the two caches.

contrl(0)	--	Opcode = 255
contrl(1)	--	Number of input vertices = 0
contrl(3)	--	Length of intin array = 1
contrl(6)	--	Device handle
intin(0)	--	Which cache to inquire size from
intout(0) intout(1)	--	Size of the largest allocatable block

C BINDING

Procedure Name

vqt_cachesize(handle, which_cache, size)

Data Types

```
int     handle;
int     which_cache;
long    *size;
```

Input Arguments

```
contrl[6] = handle;
intin[0] = which_cache;
```

Output Arguments

*size = intout[0], intout[1]

Enable Bezier capabilities

This call enables the GDOS Bezier capabilities. Note that while a handle is provided and the associated device driver is called, the GDOS Bezier extension is enabled for all devices when this call is made.

Input

contrl(0)	–	Opcode = 11
contrl(1)	–	1 (indicates ON)
contrl(2)	–	0
contrl(3)	–	0
contrl(4)	–	4
contrl(5)	–	13
contrl(6)	–	Device handle

Output

intout(0) – retval

Maximum Bezier depth. a measure of the smoothness of the curve. The value. which can range from 0 to 7, is an exponent of 2. giving the number of line segments that make up the curve. Thus. if retval is 0. the curve is actually a straight line (one line segment) If retval is 7, the curve is made of 128 line segments

---

C BINDING

Procedure Name retval = v_bez_on(handle)

Data Types

```
int     handle;
int     retval;
```

Input Arguments handle = contrl[6]

Output Arguments

retval = intout[0]

Disable Bezier capabilities

This call disables the GDOS Bezier capabilities. Any memory allocated by the GDOS for Bezier-generated polygons is released at this time. (See V_SET_APP_BUFF for memory allocation information.)

Input

contrl(0)	—	Opcode = 11
contrl(1)	—	0 (indicates OFF)
contrl(2)	—	0
contrl(3)	—	0
contrl(4)	—	0
contrl(5)	—	13
contrl(6)	—	Device handle

C BINDING

Procedure Name v_bez_off(handle)

Data Types

```
int     handle;
```

Input Arguments handle = contrl[6]

Reserve Bezier workspace

This call reserves memory for use by GDOS extensions to produce Bezier curves. When the application makes Bezier calls, the buffer set aside by this call holds the polygon generated from the Bezier anchor and direction points. When not making Bezier calls, the application has free access to this buffer. When the application exits, it must pass a zero offset, segment, and size to disable further use of this buffer and prevent accidental use of this memory. In the absence of this call, GDOS allocates its own buffer when a v_bez() or v_bez_fill() call is made. The size of this default buffer is 8K. The size of the buffer that you pass in should vary according to the complexity of the Bezier - typically around 9K bytes.

Input

contrl(0)	–	Opcode = -1
contrl(1)	–	0
contrl(2)	–	0
contrl(3)	–	0
contrl(4)	–	0
contrl(5)	–	6
contrl(6)	–	Device handle
intin(0)	–	offset to buffer (first two bytes of **address**)
intin(1)	–	segment address of buffer (last two bytes of **address**)
intin(2)	–	number of paragraphs available (each paragraph = 16 bytes)

C BINDING

Procedure Name v_set_app_buff(&address, nparagraphs)

Data Types

```
long    address;
int     nparagraphs;
```

Input Arguments nparagraphs = intin[2]

Output Arguments

address = intin[0][1]

Output Bezier

This call draws an unfilled Bezier on the specified device.

Input

contrl(0)	—	Opcode = 6
contrl(1)	—	number of vertices
contrl(2)	—	2
contrl(3)	—	(no. of vertices + 1)/2
contrl(4)	—	6
contrl(5)	—	13
contrl(6)	—	Device handle
intin(0)	—	array of vertex-type flags bit 0 = 1 first point in a 4-point Bezier segment (a curve- the four points are two anchor points and two direction points) bit 0 = 0 begins a polyline unless the previous byte's bit 0 indicates the beginning of a bezier (Note: The last point of a bezier segment can be the first point of the next segment.) bit 1 = 1 jump point - a point connecting two regions without drawing a line between them ("move to here" instead of "draw to here")
ptsin(0)	—	array of vertices

Output

intout(0)	—	no. of points in resulting polygon
intout(1)	—	no. of moves in resulting polygon
intout(2)	—	reserved
intout(3)	—	reserved
intout(4)	—	reserved
intout(5)	—	reserved
ptsout(0)	—	minimum x extent of rectangle ("bounding box") surrounding the curve
ptsout(1)	—	minimum y extent of bounding box
ptsout(2)	—	maximum x extent of bounding box
ptsout(3)	—	maximum y extent of bounding box

C BINDING

Procedure Name v_bez(handle, count, xyarr, bezarr, extent, totpts, totmoves)

Data Types

```
int     handle;
int     count, *xyarr, extent[4];
```

```
char    *bezarr;
int     *totpts, *totmoves;
```

Input Arguments

```
handle = contrl[6]
count = contrl[1]
xyarr = ptsin[0]
bezarr = intin[0]
```

Output Arguments

```
totpts = intout[0];
totmoves = intout[1];
extent[0] = ptsout[0];
extent[1] = ptsout[1];
extent[2] = ptsout[2];
extent[3] = ptsout[3];
```

Output Filled Bezier

This call draws a filled Bezier on the specified device.

Input

contrl(0)	–	Opcode = 9
contrl(1)	–	number of vertices
contrl(2)	–	2
contrl(3)	–	(no. of vertices + 1)/2
contrl(4)	–	6
contrl(5)	–	13
contrl(6)	–	Device handle
intin(0)	–	array of vertex-type flags bit 0 = 1 first point in a 4-point Bezier segment (a curve- the four points are two anchor points and two direction points) bit 0 = 0 begins a polyline unless the previous byte's bit 0 indicates the beginning of a bezier (Note: The last point of a bezier segment can be the first point of the next segment.) bit 1 = 1 jump point - a point connecting two regions without drawing a line between them ("move to here" instead of "draw to here")
ptsin(0)	–	array of vertices

Output

intout(0)	–	no. of points in resulting polygon
intout(1)	–	no. of moves in resulting polygon
intout(2)	–	reserved
intout(3)	–	reserved
intout(4)	–	reserved
intout(5)	–	reserved
ptsout(0)	–	minimum x extent of rectangle ("bounding box") surrounding the curve
ptsout(1)	–	minimum y extent of bounding box
ptsout(2)	–	maximum x extent of bounding box
ptsout(3)	–	maximum y extent of bounding box

C BINDING

Procedure Name v_bez_fill(handle, count, xyarr, bezarr, extent, totpts, totmoves)

Data Types

int handle;

```
int     count, *xyarr, *extent;
char    *bezarr;
int     *totpts, *totmoves;
```

Input Arguments

handle = contrl[6]
count = contrl[1]
xyarr = ptsin[0]
bezarr = intin[0]

Output Arguments

totpts = intout[0];
totmoves = intout[1];
extent[0] = ptsout[0];
extent[1] = ptsout[1];
extent[2] = ptsout[2];
extent[3] = ptsout[3];